Repairing and Mechanising the JavaScript Relaxed Memory Model

Conrad Watt
University of Cambridge
UK
conrad.watt@cl.cam.ac.uk

Christopher Pulte
University of Cambridge
UK
christopher.pulte@cl.cam.ac.uk

Anton Podkopaev
HSE / MPI-SWS
Russia / Germany
podkopaev@mpi-sws.org

Guillaume Barbier
ENS Rennes
France
guillaume.barbier@ens-rennes.fr

Stephen Dolan
University of Cambridge
UK
stephen.dolan@cl.cam.ac.uk

Shaked Flur
University of Cambridge*
UK
shaked.flur@cl.cam.ac.uk

Jean Pichon-Pharabod
University of Cambridge
UK
jean.pichon@cl.cam.ac.uk

Shu-yu Guo
Bloomberg LP*
USA
shu@rfrn.org

Abstract

Modern JavaScript includes the SharedArrayBuffer feature, which provides access to true shared memory concurrency. SharedArrayBuffers are simple linear buffers of bytes, and the JavaScript specification defines an axiomatic relaxed memory model to describe their behaviour. While this model is heavily based on the C/C++11 model, it diverges in some key areas. JavaScript chooses to give a well-defined semantics to data-races, unlike the "undefined behaviour" of C/C++11. Moreover, the JavaScript model is *mixed-size*. This means that its accesses are not to discrete locations, but to (possibly overlapping) ranges of bytes.

We show that the model, in violation of the design intention, does not support a compilation scheme to ARMv8 which is used in practice. We propose a correction, which also incorporates a previously proposed fix for a failure of the model to provide Sequential Consistency of Data-Race-Free programs (SC-DRF), an important correctness condition. We use model checking, in Alloy, to generate small counter-examples for these deficiencies, and investigate our correction. To accomplish this, we also develop a mixed-size extension to the existing ARMv8 axiomatic model.

Guided by our Alloy experimentation, we mechanise (in Coq) the JavaScript model (corrected and uncorrected), our ARMv8 model, and, for the corrected JavaScript model, a "model-internal" SC-DRF proof and a compilation scheme correctness proof to ARMv8. In addition, we investigate a non-mixed-size subset of the corrected JavaScript model, and give proofs of compilation correctness for this subset to x86-TSO, Power, RISC-V, ARMv7, and (again) ARMv8, via the Intermediate Memory Model (IMM).

As a result of our work, the JavaScript standards body (ECMA TC39) will include fixes for both issues in an upcoming edition of the specification.

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; • **General and reference** → *Verification*.

Keywords: Alloy, ARMv8, Coq, weak memory, Web worker

1 Introduction

JavaScript is widely publicised as a "single-threaded language" [39], with asynchronously dispatched events processed by a single event loop. JavaScript (JS) allows the use of threads, called *Web Workers*, for parallel computation, but until recently these were not allowed to share access to mutable state, and inter-thread communication was restricted purely to message-passing [11]. However, a new feature of JavaScript, SharedArrayBuffer, allows true concurrent access to a low-level shared resource [18]. SharedArrayBuffers are simple linear mutable byte buffers, and, unlike other JavaScript objects, references to the same SharedArrayBuffer may be held by multiple Web Workers simultaneously.

SharedArrayBuffers were originally specified and implemented by all major browsers in 2017, but were disabled shortly after due to concerns about Spectre and Meltdown. Now that mitigations have been developed, the feature has been re-introduced into the web ecosystem [15].

SharedArrayBuffers have several important uses. They are the only mechanism in JavaScript for true shared-memory concurrency. C++ is often compiled to asm.js [28], a fast JavaScript subset, for use on the web, and concurrent C++

* At the time the work was done.

objects must be allocated on SharedArrayBuffers in the compiled program. SharedArrayBuffers also provide the mechanism by which JavaScript may interoperate with concurrent WebAssembly programs [27].

The JavaScript specification must define the concurrent behaviours that are possible when the same SharedArrayBuffer is accessed concurrently by multiple Web Workers; this is done through a *relaxed memory model*. The relaxed memory model of JavaScript was designed to conform to an ambitious set of requirements [20]:

Mixed-size accesses. JavaScript's concurrency is *mixed-size*, in the sense of Flur et al. [22]: accesses are not to individual, discrete locations, but instead to ranges of byte locations, which may overlap with each other.

Mixed atomic and non-atomic accesses. JavaScript has no concept of an "atomic location", so atomic and non-atomic accesses may be arbitrarily combined on the same location. JavaScript has only one type of atomic access, **SeqCst**, while C/C++11 also has so-called "low-level atomics" such as release/acquire.

No undefined behaviour. The JavaScript language does not have a concept of undefined behaviour, so all programs must have behaviour defined by the standard. This remains true even in the presence of data-races, although the defined behaviour is then extremely weak.

C++-compatible compilation. JavaScript atomic accesses are to use the same compilation scheme as C++ SC-atomic accesses [20, p. 17].

SC-DRF. Programs that are free of data races must have sequentially consistent behaviour [20, p. 8].

Whatever JavaScript's general reputation, it should be emphasised that its current specification is particularly rigorous. Because of its ubiquity on the Web and the large number of language implementers, great care is taken to ensure that its features are precisely defined.[1] In particular, its relaxed memory model is defined using an unambiguous, semi-formal pseudocode, which takes inspiration from previous formalisations of the memory models of C/C++11 [9] and Java [34]. In addition, SharedArrayBuffers have been carefully designed to participate as little as possible in JavaScript's complicated object inheritance model, effectively allowing us to reason about them in isolation.

[1] JavaScript's least intuitive behaviours arise not out of failures of its current specification process, but out of a requirement to be backwards-compatible with earlier versions of the language. Often, this means specifying a strange behaviour for legacy reasons.

1.1 Mixed-Size

In some respects, the JavaScript relaxed memory model is similar to that of C/C++ [9], sharing an axiomatic nature and several core definitions and relations. However, the mixed-size nature of the JavaScript model is a substantial complication in its verification. Unlike what prior work typically assumes, memory accesses can be of different sizes (byte-widths); hence, two accesses may overlap without having the exact same "footprint", adding another dimension of complexity to the model, and limiting our ability to make use of prior work in which this is assumed not to occur.

To the best of our knowledge, only a single previous work, Flur et al. [22], deals with the formal verification of compilation of a mixed-size relaxed memory model. Flur et al. concentrate mainly on architecture-level mixed-size behaviours, proposing operational mixed-size models for the ARMv8 and POWER architectures. Their work describes an extension to an existing formalisation of C/C++11, adding mixed-size non-atomics, and gives a sketch hand-proof that the resulting model can be correctly compiled to POWER, acknowledging that fuller verification is an open problem. Even this mixed-size C/C++11 model only allows mixed-size non-atomics, on which a data race leads to undefined behaviour. The JavaScript memory model, in contrast, must give well-defined behaviour even in the case of data races between partially-overlapping accesses.

Two other papers deal with mixed-size models. Watt et al. [52] describe a memory model for WebAssembly that is closely related to the JavaScript memory model. The authors do not attempt a proof of correctness of compilation, again declaring it as an open problem. They report on a deficiency in the JavaScript memory model which we investigate further as part of this work. Finally, the EMME tool [37] represents an earlier investigation of JavaScript's memory model using the Alloy model checker [29]. This tool is engineered primarily as a test oracle, and the work does not attempt compilation scheme verification. We also make use of Alloy during this work, but primarily for compilation scheme investigation, in the style of Memalloy [53]. We discuss differences in our approaches in §8.

1.2 Our Contributions

ARMv8 compilation scheme failure. We have discovered that, on ARMv8, compiling JavaScript atomics by following the standard compilation scheme for C++ SC atomics allows behaviours which violate the guarantees of the JavaScript memory model. This compilation scheme is already implemented in Google's V8 JavaScript engine [13], and we are able to observe the violating behaviour experimentally through Web browsers on real hardware. After consultation with implementers and ECMA TC39, the JavaScript standards body, we concluded that the JavaScript memory model should be weakened in order to support this scheme (§3.1).

Fixing the JS specification. The JavaScript model also fails to guarantee Sequential Consistency for Data-Race-Free programs (SC-DRF), a crucial correctness condition [2, 12, 23, 31]. This was discovered in previous work [52], which proposed a strengthening of the model to restore SC-DRF, but did not verify that the strengthened condition is supported by existing compilation schemes. We integrate our fix for the ARMv8 compilation issue with this previous proposal, obtaining a combined fix to the JavaScript model (§3.2). On the strength of our model checking and verification work (detailed below), this combined fix was adopted by the standards body for inclusion in an upcoming edition of the standard.

Alloy model checking. Using the Alloy model checker [29], we can automatically find counter-examples exemplifying the above two deficiencies, following the approach of Memalloy [53], but here, for the first time, applied in a mixed-size context. We also use Alloy to inform our Coq compilation scheme verification of the revised model, by essentially model checking (up to a bound) the main construction used by that proof (of JavaScript-allowed executions from ARMv8-allowed executions).

Mixed-size axiomatic ARMv8 model. To enable verifying the compilation scheme, we define, first in Alloy, then in Coq, a novel mixed-size ARMv8 axiomatic model, as a generalisation of ARM's axiomatic reference model, and validate it with respect to Flat [22, 43], a well-tested mixed-size operational model for ARMv8 (§4).

Coq proofs. We mechanise the JavaScript and mixed-size ARMv8 models in Coq, and give a proof of compilation scheme correctness, and the "model-internal" SC-DRF property (see §3.2) for the revised model. We investigate the (subtle) circumstances under which the mixed-size model may be reduced to an equivalent non-mixed-size (hereafter "uni-size") model. We define a uni-size subset of the JavaScript model and prove, in Coq, compilation scheme correctness for this subset to x86-TSO, Power, RISC-V, ARMv7, and (again) ARMv8 via the IMM model [38, 42] (§6).

Thread suspension specification. Looking beyond memory accesses, JavaScript also defines thread suspension operations: "`Atomics.wait`", conditionally blocking a thread, and "`Atomics.notify`", unblocking waiting threads. The synchronization guarantees of these operations were not integrated into the formal model, leading to ambiguities which we correct (§7).

Our artefacts are distributed as supplemental material [51].

1.3 Non-contributions

C/C++11-style "out-of-thin-air" executions [7] are admitted by the JavaScript model for certain programs with racing non-atomics [52]. We acknowledge this is a deficiency of the model, but we do not attempt to solve this long-standing problem here; proposed solutions for C/C++11 involve either performance sacrifices [31] or the adoption of a radically different model [30].

2 JavaScript's Shared Memory

As previously mentioned, JavaScript allows threads to concurrently access SharedArrayBuffer objects, which are simply zero-initialised raw buffers of bytes.

To access a SharedArrayBuffer, the programmer must declare a special wrapper around it called a *typed array*. Every typed array has a *width*, which is the number of consecutive bytes in the underlying SharedArrayBuffer that a single load or store on the typed array will access. Fig. 1 depicts a simple two-threaded program that initially declares a single SharedArrayBuffer of 1024 bytes, and wraps it in a typed array with a width of 32 bits (4 bytes). The two threads then perform a simple *message-passing* procedure, which we use to illustrate JavaScript's two access modes. Thread 0 writes the value 1 to location 0 (the *message*) using a standard non-atomic access (note that this corresponds to bytes 0-3 of the underlying SharedArrayBuffer). It then writes 1 to location 1 (the *flag*, at bytes 4-7) using a sequentially consistent atomic access. Thread 1 reads location 1 atomically and then, only if it observes thread 0's write, reads location 0.

The paired atomic read/write on location 1 give strong ordering guarantees. Two possible outcomes are allowed: either `r0 = 5 ∧ r1 = 3` or `r0 = 0`. In particular, the outcome `r0 = 5 ∧ r1 = 0`, where the flag is observed as set but the message is not received, is not allowed. However, if either of the two atomic operations are replaced with non-atomics, then the outcome `r0 = 5 ∧ r1 = 0` *can* be experimentally observed. This is an example of *relaxed memory behaviour*.

As a convention, when we depict JavaScript code fragments, all accesses will be assumed to be to 32-bit typed arrays unless otherwise stated. A single SharedArrayBuffer may be wrapped by multiple typed arrays of different widths, leading to mixed-size behaviours, where accesses partially overlap with each other.

JavaScript also provides a low-level mechanism for manipulating SharedArrayBuffers called a DataView. DataViews only offer non-atomic operations, which may uniquely be *unaligned*. DataViews are far less commonly-used than typed arrays; they have historically been avoided due to performance problems [25], with Emscripten [54] (a key JavaScript-producing toolchain) generating code that uses typed arrays exclusively.

```
x = new Int32Array(new SharedArrayBuffer[1024]);
```

Thread 0	Thread 1
`x[0] = 3;`	`r0 = Atomics.load(x,1);`
`Atomics.store(x,1,5);`	`if (r0 == 5) {`
	`  r1 = x[0];`
	`}`

Figure 1. A simple JavaScript program.

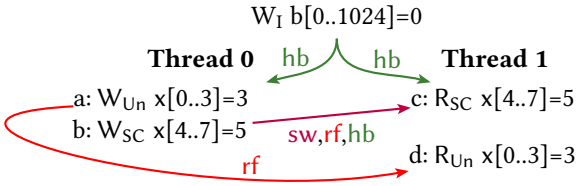


Figure 2. A candidate execution for Fig. 1.

Most of our results cover DavaView-generated unaligned accesses, with the exception of our Coq compilation scheme correctness proof (§6.2), which handles only the aligned (but still possibly mixed-size) accesses generated by typed arrays.

We now define the JavaScript relaxed memory model as it appears in the latest (10^{th}) edition of the specification [19]. This model describes the range of relaxed memory behaviours that a JavaScript program is allowed to exhibit. In the course of this paper, we will present and discuss alterations to the model that have been accepted for inclusion in a future edition, which fix several deficiencies in the model as presented here.

2.1 Thread-Local and Axiomatic Semantics

JavaScript, like C++, has an axiomatic relaxed memory model. This means that the allowed concurrent behaviours are stated as whole-execution axiomatic constraints. A program's semantics is defined in two layers. First, an operational thread-local semantics of the language describes how each thread executes. However, the values of read operations which access shared memory locations are not concretely determined at this stage. Instead, each operation will arbitrarily and non-deterministically pick a value to continue execution with, and generate an *event* which records the choice made, and the location accessed. Similarly, write operations to shared memory do not concretely mutate program state, but instead generate an event recording what was written, and where.

Given a complete execution of all threads at this thread-local level, the specification defines a structure called a *candidate execution*.[2] Intuitively, a candidate execution represents an execution that is consistent with the language's sequential semantics. It contains the set of all events generated by the thread-local semantics, together with a possible justification for the arbitrarily chosen read values, which must be checked further.

One part of this justification is a "reads-from" relation which must link every read event (with an arbitrarily picked value) to a write event which writes the value that was picked. Other relations are included in the justification which enforce ordering constraints on the shape of the reads-from relation (representing, for example, inter-thread synchronization).

[2] In C++, this is known as a *pre-execution*.

The language defines a second layer of axiomatic constraints over candidate executions, the *axiomatic memory model*, which classifies candidate executions as either valid or invalid. It is required by the specification that any concretely observable execution must correspond to some valid candidate execution. In this way, the memory model determines which read/write values chosen at the level of the thread-local semantics are permitted to be observed.

2.2 Candidate Executions

Candidate executions are formally specified in Fig. 3. A candidate execution for a given thread-local execution consists of the set of events **evs**, and the relations over this set **sequenced-before**, **additional-synchronizes-with**, **reads-byte-from**, and **total-order**, which form the potential justification.

Our notation mainly follows that of the formal C/C++11 memory model [9]; we treat relations between events as sets of tuples, and make use of standard notation from relational algebra to manipulate them. For example, the transitive closure of a binary relation rel is given as rel^+, and its inverse as rel^{-1}. For binary relations, we use an infix notation to indicate membership i.e. $A \ \text{rel} \ B \equiv \langle A, B \rangle \in \text{rel}$.

The candidate execution's components evs, sequenced-before, and additional-synchronizes-with are all precisely determined from the thread-local execution.

Events generated by the thread-local execution are contained in **evs**. Each event records its mode, which may be either Sequentially Consistent (atomic), unordered (non-atomic), or a specially distinguished initializing write, and the locations that it accesses: the combination of the block, representing the address of an individual SharedArrayBuffer; the index, representing the starting position of the access within the SharedArrayBuffer; and the reads and writes fields, representing the list of bytes the event read or wrote (respectively) as determined by the thread-local semantics. Moreover, the thread-local semantics marks certain events as *tearfree*, which will be explained later. The block component's main function is merely to ensure that accesses to different SharedArrayBuffers are treated as having disjoint ranges by construction. Hence, throughout this paper we will usually work with the assumption that all accesses in a candidate execution are to the same block.

$$
\begin{array}{lcl}
\textit{mode} & ::= & \textbf{Unordered} \;/\!/\; \textbf{Un} \mid \textbf{SeqCst} \;/\!/\; \textbf{SC} \mid \textbf{Init} \;/\!/\; \textbf{I} \\
\textit{event} & ::= & \{\, \text{ord} : \textit{mode} \\
 & & \;\; \text{block} : \textit{addr} \\
 & & \;\; \text{index} : \textit{nat} \\
 & & \;\; \text{reads} : \textit{list byte} \\
 & & \;\; \text{writes} : \textit{list byte} \\
 & & \;\; \text{tearfree} : \textit{bool} \,\}
\end{array}
\qquad
\begin{array}{lcl}
\textit{addr} & ::= & \alpha \ldots \text{an infinite set of abstract names} \\
\textit{candidate\_execution} & ::= & \{\, \text{evs} : \textit{set event} \\
 & & \;\; \text{sequenced-before} \;/\!/\; \textbf{sb} : \textit{set}\,(\textit{event} \times \textit{event}) \\
 & & \;\; \text{additional-synchronizes-with} \;/\!/\; \textbf{asw} : \textit{set}\,(\textit{event} \times \textit{event}) \\
 & & \;\; \text{reads-byte-from} \;/\!/\; \textbf{rbf} : \textit{set}\,(\textit{nat} \times \textit{event} \times \textit{event}) \\
 & & \;\; \text{total-order} \;/\!/\; \textbf{tot} : \textit{set}\,(\textit{event} \times \textit{event}) \,\}
\end{array}
$$

$$
\begin{array}{rcl}
\text{range}_r(E : \textit{event}) & \triangleq & [\, E.\text{index} \ldots E.\text{index} + |E.\text{reads}| \,) \\
\text{range}_w(E : \textit{event}) & \triangleq & [\, E.\text{index} \ldots E.\text{index} + |E.\text{writes}| \,) \\
\text{range}(E : \textit{event}) & \triangleq & \text{range}_r(E) \cup \text{range}_w(E)
\end{array}
\qquad
\begin{array}{rcl}
\text{write}(E : \textit{event}) & \triangleq & (E.\text{writes} \neq [\,]) \\
\text{overlap}(E_1, E_2 : \textit{event}) & \triangleq & E_1.\text{block} = E_2.\text{block} \wedge \\
 & & \text{range}(E_1) \cap \text{range}(E_2) \neq \emptyset
\end{array}
$$

Derived relations *(wrt. a candidate execution)*

$$
\begin{array}{l}
\text{reads-from} \;/\!/\; \textbf{rf} \triangleq \\
\quad \{\langle A, B\rangle \mid \exists k.\, \langle k, A, B\rangle \in \text{reads-byte-from}\} \\
\text{happens-before} \;/\!/\; \textbf{hb} \triangleq \\
\quad \left(\begin{array}{l}\text{sequenced-before} \cup \text{synchronizes-with} \cup \\ \{\, \langle A, B\rangle \mid A.\text{ord} = \textbf{Init} \wedge \text{overlap}(A, B) \,\}\end{array}\right)^{+}
\end{array}
$$

$$
\begin{array}{l}
\text{synchronizes-with} \;/\!/\; \textbf{sw} \triangleq \\
\quad \left\{ \langle A, B\rangle \;\middle|\; \begin{array}{l} A \text{ reads-from } B \wedge B.\text{ord} = \textbf{SeqCst} \wedge \\ \left(\begin{array}{l}(\text{range}_w(A) = \text{range}_r(B) \;\wedge A.\text{ord} = \textbf{SeqCst}) \vee \\ (\forall C.\; C \text{ reads-from } B \longrightarrow C.\text{ord} = \textbf{Init})\end{array}\right)\end{array} \right\} \\
\quad \cup \text{additional-synchronizes-with}
\end{array}
$$

Figure 3. JavaScript Candidate Execution. We introduce short names for some relations after the "$/\!/$".

The **sequenced-before** component is an intra-thread relation between events that records their order in the control-flow unfolding of the execution. It ensures that events that occur sequentially in the same thread are strongly ordered with respect to each other.

The relation **additional-synchronizes-with** records places where the thread-local semantics' action implies strong inter-thread ordering in the memory model. For example, when a parent thread creates a child thread, the thread-local semantics has an additional-synchronizes-with edge from the parent to the child. This edge will ensure that all previous accesses by the parent are visible to the child.

In addition, the candidate execution contains two relations which do not merely arise from the thread-local semantics. The **reads-byte-from** component represents a possible justification for the values of read events, by relating them to write events in the execution. It is defined in a byte-wise manner; each byte location of a multi-byte read event is related to a write event on that location, and each byte may be justified by a different write. Therefore $\langle k, E_w, E_r\rangle \in$ reads-byte-from means the event E_r reads the value of E_w at byte index k.

Finally, the **total-order** component records some total order over all events. Sequentially consistent atomic operations must obey certain restrictions about where they can appear in this total order, resulting in stronger guarantees about their behaviour.

The relations reads-byte-from and total-order are arbitrarily picked when constructing the candidate execution, subject to certain intuitive well-formedness conditions (reads-byte-from must associate read events to write events with the same byte values, total-order must be a strict total order on events) which we define explicitly in a supplementary appendix [51]. At the programmer level, an execution is only observable if, for some choice of reads-byte-from and total-order, a candidate execution exists which is allowed by the memory model.

Fig. 3 also defines three derived relations: intuitively, the reads-from relation recovers a C/C++11-style event-to-event definition from the reads-byte-from relation, by projecting away the byte index component[3]; the synchronizes-with relation records the extra synchronization guarantees made by **SeqCst** atomics; the happens-before relation is the transitive closure of different ordering constraints.

Throughout the paper, we will give graphs representing (fragments of) candidate executions. We give a simple example in Fig. 2, which depicts a valid candidate execution, including relevant derived relations, for Fig, 1, which justifies the outcome `r0 = 5 ∧ r1 = 3`. Each event in the candidate execution corresponds to a load/store operation performed by the thread-local semantics. Some relation edges are elided where irrelevant (for example, the precise choice of **tot** is not interesting in this example) or otherwise obvious (**sb** is trivial from the program layout).

2.3 Valid Candidate Executions

As discussed, an execution is allowed by the specification if it is possible to pick reads-byte-from and total-order relations (in C/C++11 called an *execution witness* [9]) such that the resulting candidate execution is valid. Validity of a candidate execution is defined in Fig. 4.

[3] By convention, and in common with C/C++11, we make the write the *left* component of the reads-from relation.

Happens-Before Consistency (1):
$\text{happens-before} \subseteq \text{total-order}$

Happens-Before Consistency (2):
$\forall E_w E_r.\ E_w \text{ reads-from } E_r \longrightarrow \neg(E_r \text{ happens-before } E_w)$

Happens-Before Consistency (3):
$\forall \langle k, E_w, E_r \rangle \in \text{reads-byte-from}.$
$\quad \nexists E'_w.\ (E_w \text{ happens-before } E'_w) \wedge$
$\quad\quad (E'_w \text{ happens-before } E_r) \wedge k \in \text{range}_w(E'_w)$

Tear-Free Reads:
$\forall E_r.\ E_r.\text{tearfree} \longrightarrow$
$\left|\left\{ E_w \;\middle|\; E_w \text{ reads-from } E_r \wedge E_w.\text{tearfree} \wedge \text{range}_w(E_w) = \text{range}_r(E_r) \right\}\right| \leq 1$

Sequentially Consistent Atomics (first attempt):
$\forall E_w E_r.\ E_w \text{ synchronizes-with } E_r \longrightarrow \nexists E'_w.\ (E_w \text{ total-order } E'_w) \wedge (E'_w \text{ total-order } E_r) \wedge \text{range}_w(E'_w) = \text{range}_r(E_r)$

Figure 4. Candidate execution validity as defined by the latest JavaScript specification.

Happens-Before Consistency (1-3). An edge in happens-before implies a strong ordering constraint in the model. Rule (1) states that the total order **tot** must contain **hb**. The other rules ensure that reads do not, under any circumstances, observe writes in a way that is inconsistent with happens-before; (2) a read cannot be happens-before a write it reads from; (3) a read E_r cannot read "stale" bytes from a write E_w if there is a "newer" write E'_w according to happens-before.

Tear-Free Reads. This rule provides extra guarantees on the behaviour of events marked as tearfree. A *tearing* event (one that is not tearfree) represents an access which be may be observed as a series of smaller independent accesses. One example would be a 64-bit access implemented on a 32-bit machine as a pair of 32-bit accesses.

For events declared as tearfree, this rule guarantees that tearfree reads will never read from more than one tearfree write of the same size and alignment. That is, it will not observe an interleaving of bytes from multiple tearfree writes.

Sequentially Consistent Atomics. Finally, the SC Atomics rule is intended to further restrict **SeqCst** atomics, so that **SeqCst** reads observing **SeqCst** writes obey the total-order. Note that it is still possible for more relaxed behaviour to occur if an **Unordered** access is intermingled with **SeqCst** ones.

As we will show in §3, this last condition must be rewritten to correct deficiencies in the model.

JavaScript's specification describes the memory model in a precise, semi-formal pseudocode. When rendering the model in logic, it is convenient for us to make some changes in presentation that do not affect the model. These are discussed in a supplemental appendix [51].

3 Corrected Model Deficiencies

The current JavaScript concurrency model contains two major deficiencies that will be discussed in the following, along with our proposed alterations to the model. These proposals have now been accepted by the JavaScript committee for inclusion in an upcoming edition of the standard.

Throughout this paper, we will work with a restricted fragment of the JavaScript language, consisting only of programs with a fixed number of threads, in which each thread has only shared memory accesses and simple control-flow, and where the program contains an already-initialised SharedArrayBuffer (potentially wrapped by multiple typed arrays). We assume that the initialisation is done before all other accesses, since the concurrent behaviour of memory allocation relies on the (relaxed) behaviour of dynamic allocation involving OS calls, which is beyond the scope of this work to reason about. This fragment is sufficient to exhibit all the deficiencies discussed in the next sections, and later, verify their absence in the revised model incorporating our fixes. This language fragment is the JavaScript equivalent to the fragment of C/C++11 considered by previous work such as [42] and [31], with the additional complication that our accesses may be mixed-size.

After discussing the details of the deficiencies of the current JavaScript model, we show our use of Alloy for generating counter-examples for the original model in §5, and we detail our Coq verification of the corrected model in §6. We also discover problems with another feature of the language, relating to thread suspension, which we describe in §7.

3.1 ARMv8 Compilation

The ARMv8 architecture provides the Load Acquire (`ldar`) and Store Release (`stlr`) instructions, memory access instructions with certain thread-local ordering guarantees, which are also intended as compilation targets for C/C++ sequentially consistent atomics (`memory_order_seq_cst`). It was intended that the JavaScript model should support this compilation scheme, which is implemented in at least one Web browser (Chrome). We have identified, however, that the current JavaScript memory model, as presented in Fig. 4, is incompatible with this compilation scheme. The key issue is the **Sequentially Consistent Atomics** condition, which, in addition to restricting **SC** accesses, also restricts **Un** accesses by disallowing executions of the shape shown below.

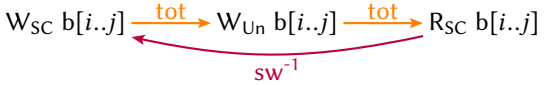
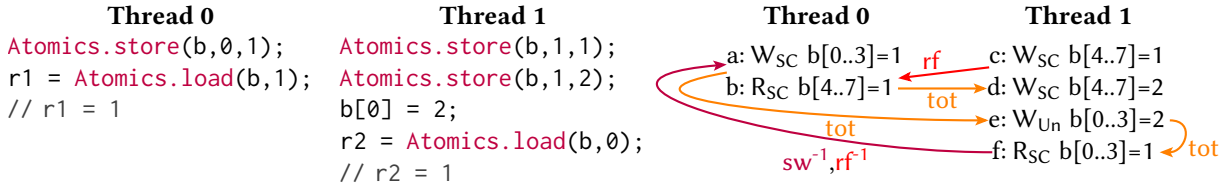
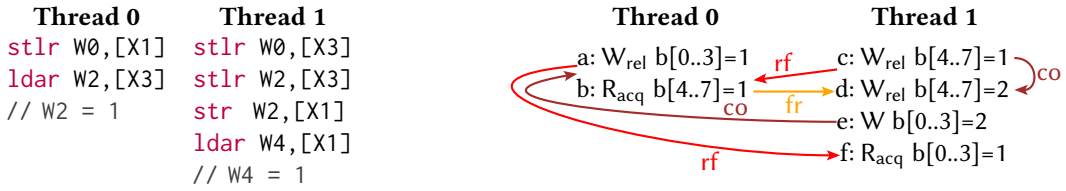


Figure 5. Forbidden by **SC Atomics (first attempt)**.



(a) An outcome forbidden by JavaScript. Note the shape of Fig. 5 appearing between events a, e, and f.



(b) When the program is compiled to ARMv8, the outcome is allowed.

Figure 6. A JavaScript program which violates the memory model when compiled to ARMv8.

It is possible to craft a program that produces a particular output only in a candidate execution that contains this shape forbidden by JavaScript. Nevertheless, the behaviour is observable on ARMv8 when using the compilation scheme that maps **Un** accesses to bare ARMv8 accesses and **SC** accesses to ARMv8 release/acquire accesses. Such a program is shown in Fig. 6. The candidate execution of Fig. 6a is (in the unfixed JavaScript model) forbidden because it includes the shape of Fig. 5. Note that no other candidate execution can make this output observable, since alternative configurations of edges are also forbidden by the memory model.

In particular, because the event (b) reads 1, there must be a **tot** edge from (b) to the write (d). If the edge were the other way around, (b) would not be allowed to read 1, and could only read 2 from (d), since reading from (c) would be forbidden by the **Sequentially Consistent Atomics** rule. Therefore the **tot** edge from (a) to (e) is also fixed, because of **tot**'s transitivity and the fact that **hb** $\in$ **tot**.

We originally discovered a larger counter-example by hand; this small counter-example was found automatically as part of our Alloy model-checking efforts, as detailed in §5. We have confirmed that the corresponding execution is architecturally allowed in ARMv8 for the compiled program, by running it in the two existing executable concurrency models for ARMv8 [17, 43]. We are also able to observe this execution experimentally, with the caveat that we must use WebAssembly to force efficient compilation (see §3.3).

Proposed Fix. We propose a weakening of the JavaScript model which permits this ARMv8 compilation scheme: weakening the **Sequentially Consistent Atomics** condition of Fig. 4 as follows:

SC Atomics (second attempt):
$\forall E_w E_r.\ E_w$ synchronizes-with $E_r \longrightarrow$
$\quad \nexists E'_w.\ E'_w$.ord = **SeqCst** $\wedge$ (E_w total-order E'_w) $\wedge$
$\qquad (E'_w$ total-order $E_r) \wedge \text{range}_w(E'_w) = \text{range}_r(E_r)$

This means that shapes like Fig. 5 are no longer forbidden. Intuitively, the original condition was putting an ordering constraint on **Un** accesses that were part of a data-race (see §3.2), by forcing them to have a certain position in **tot**, even when that position was not enforced by **hb**. The ARMv8 `ldar/stlr` instructions were designed to support C/C++11 atomics where such a data-race would be undefined behaviour, and they do not provide the guarantees necessary for (uncorrected) JavaScript's **Un** ordering in the racy case.

3.2 SC-DRF

Watt et al. [52] identified that the JavaScript model does not provide Sequential Consistency (SC) of Data-Race-Free programs (SC-DRF), an important correctness condition of the relaxed memory model that JavaScript intends to provide. After a discussion of JavaScript's choice of SC-DRF definition, we detail JavaScript's violation of this SC-DRF property and integrate their proposed correction with our ARMv8 fix, for subsequent verification.

Informally, the SC-DRF property says that a data-race-free program will only give rise to SC results, i.e. results corresponding to a sequential interleaving of its accesses [32]. This is an important property because it allows programmers to reason about their software under a simpler semantics: so long as they ensure their programs are data-race-free, they can program according to the simpler SC model.

Discussion of SC-DRF definition. The JavaScript standard explicitly specifies the SC-DRF property it intends to provide. Their specification of SC-DRF is analogous to the statement of the property given in the C++11 standard, and used by Batty et al. [8]: informally, two JavaScript accesses are considered to data-race if they overlap, at least one of them is a write, they are not both same-range **SC** atomics, and the two accesses are not ordered by **hb**. The formal definition is given in Fig. 7.

Data-Race: (for two events A and B in a given CE)
$(A.\text{ord} = \textbf{Un} \vee B.\text{ord} = \textbf{Un} \vee \text{range}(A) \neq \text{range}(B)) \wedge$
$\text{overlap}(A, B) \wedge (\text{write}(A) \vee \text{write}(B)) \wedge$
$\neg(A \text{ happens-before } B \vee B \text{ happens-before } A)$

Figure 7. Definition of a JavaScript data-race.

A program is then called data-race-free if it has no (JavaScript-allowed) execution containing a data-race, and the JavaScript specification says that such a data-race-free program should only have SC behaviours.

A "model-agnostic" definition of SC-DRF has since been proposed (but not adopted) for C/C++ [7], based on a simpler definition of data-race-freedom that requires the absence of data-races only in *Sequentially Consistent executions*, instead of every possible execution allowed by the model. This paper concentrates exclusively on JavaScript's (and by extension, C++11's) formulation, which we refer to as "model-internal SC-DRF" where appropriate, in order to disambiguate our verification claims.

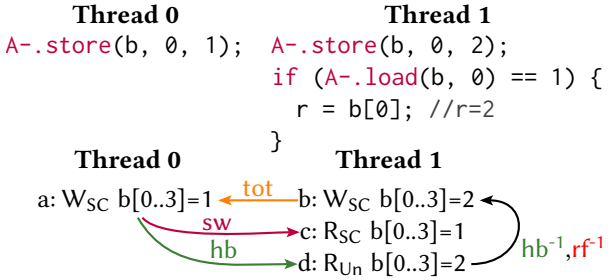


Figure 8. SC-DRF violation by JavaScript program.

JavaScript SC-DRF failure. The JavaScript specification claims that the model is SC-DRF. However, as described by Watt et al. [52], it is possible to give a counter-example: a program that is data-race-free, but nevertheless has an execution which cannot be explained as a sequential interleaving of the program's accesses.

That paper describes a 6 event, 2 (distinct) location counter-example. Using the Alloy search of §5, we are able to find a 4 event, 1 location counter-example (Fig. 8). No sequential interleaving of the program's accesses can explain why the non-atomic load of thread 1 can read 2.

Watt et al. [52] propose a strengthening of the model that would restore SC-DRF, by adding two sub-conditions to **Sequentially Consistent Atomics**. These disallow the two shapes shown in Fig. 9.

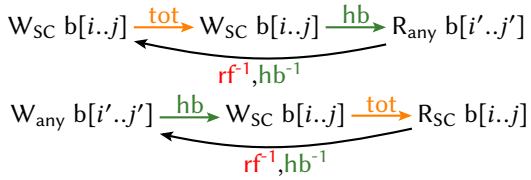
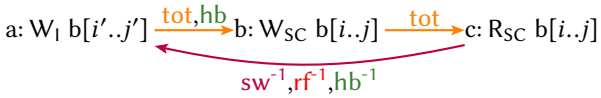


Figure 9. SC-DRF violations forbidden by the revised rule.

Combining this proposal with our ARMv8 fix, we arrive at the version of the **Sequentially Consistent Atomics** condition that we proposed to the standards committee, and which has been adopted for future inclusion (Fig. 10). This new condition is neither stronger nor weaker than the original formulation. The ARMv8 fix weakens the model, allows some previously forbidden executions. The SC-DRF fix strengthens the model, forbidding some previously allowed executions. We verify in Coq that the revised model supports the desired ARMv8 compilation scheme, and provides model-internal SC-DRF (§6).

Moreover, we identify that this condition allows another part of the model to be simplified. The model's definition of synchronizes-with includes a special case for **Init** events, ensuring that the below shape is forbidden by the **Sequentially Consistent Atomics** rule.



Note that it is always guaranteed that a **hb** b and a **hb** c by the definition of happens-before. Also, **sw** $\subseteq$ **rf**. Therefore, the revised definition of **Sequentially Consistent Atomics** already forbids a more general shape (the second shape of Fig. 9) and we can remove this special case, simplifying the definition of synchronizes-with, as shown below.

$$\text{synchronizes-with} \;//\; \textbf{sw} \triangleq \left\{ \langle A, B\rangle \;\middle|\; \begin{array}{l} A \text{ reads-from } B \wedge \text{range}_w(A) = \text{range}_r(B) \wedge \\ A.\text{ord} = B.\text{ord} = \textbf{SeqCst} \end{array} \right\} \cup \text{additional-synchronizes-with}$$

Sequentially Consistent Atomics (final):

$$\forall E_w E_r.\, E_w \text{ reads-from } E_r \;\wedge\; E_w \text{ happens-before } E_r \longrightarrow \nexists E'_w.\, E'_w.\text{ord} = \textbf{SeqCst} \;\wedge\; E_w \text{ total-order } E'_w \;\wedge\; E'_w \text{ total-order } E_r \wedge \left(\begin{array}{l} (\text{range}_w(E'_w) = \text{range}_r(E_r) \;\wedge\; E_w \text{ synchronizes-with } E_r) \\ \vee\, (\text{range}_w(E_w) = \text{range}_w(E'_w) \wedge E_w.\text{ord} = \textbf{SeqCst} \;\wedge\; E'_w \text{ happens-before } E_r) \\ \vee\, (\text{range}_w(E'_w) = \text{range}_r(E_r) \;\wedge\; E_w \text{ happens-before } E'_w \;\wedge\; E_r.\text{ord} = \textbf{SeqCst}) \end{array} \right)$$

Figure 10. The **Sequentially Consistent Atomics** rule containing all proposed fixes.

3.3 Experimental Observations

As discussed, the ARMv8 model specifies that the execution of Fig. 6a is architecturally allowed, and so potentially observable when the code is run in the V8 JavaScript engine, a component of the Chrome Web browser, that uses the release/acquire ARMv8 compilation scheme. We attempted to observe this behaviour "end-to-end", by building a website running the JavaScript fragment, but were unsuccessful. JavaScript compilation is complex, and incorporates profile-guided optimisation. We found that we could not coax the engine to generate the efficient ARMv8 code of Fig. 6. However, we can take advantage of the fact that WebAssembly's memory model (for this language fragment) is designed to be identical to JavaScript's [52]; the exact same accesses are available as WebAssembly instructions. Indeed, V8 compiles JavaScript and WebAssembly through the same backend.

The predictability of WebAssembly compilation as a proxy for perfectly optimised JavaScript was previously taken advantage of by the RIDL MDS attacks [48]. Here, instead, we use it to gain more predictability over the compilation of the litmus test. By embedding the same test, in WebAssembly, on a website, we were able to observe the problematic ARMv8 behaviour in Chrome, on the LG G Flex2 H955 phone, an Android phone with a Qualcomm Snapdragon810 SoC (quad core ARM Cortex-A57 + quad core ARM Cortex-A53). Due to the general shape of the test, we conjecture that any CPU exhibiting the `R+polp+pola` litmus test [35] should also exhibit the counter-example behaviour.

Our experimental evidence was sufficient to motivate to the JavaScript committee that this was a practical problem that needed to be addressed, as they aim for the JavaScript and WebAssembly accesses to have identical semantics.

Before detailing our Alloy-based counter-example search and model-checking of the ARMv8 compilation scheme and the SC-DRF property for the fixed model in §5, we now discuss our work on defining a mixed-size ARMv8 model.

4 ARMv8 Mixed-Size Model

In order to enable the Alloy-based counter-example search and bounded verification of §5, and the Coq compilation scheme correctness proofs of §6, we define and validate a mixed-size ARMv8 axiomatic model, as an extension of the existing ARMv8 axiomatic reference model.

The two starting points for developing the mixed-size axiomatic model are the existing Flat model [43], an operational model with mixed-size support, and ARM's reference model [17, 43], an axiomatic specification defined in herd [6], without mixed-size support. The two models are based on extensive past research on architectural concurrency for ARM (and related Power), discussion with architects, and experimental hardware testing [1, 3–6, 14, 16, 21, 22, 24, 33, 36, 43–45]. The mixed-size axiomatic model we arrive at is a generalisation of the reference axiomatic model to mixed-size programs in a way that aims to follow the Flat model's behaviour — Flat has been developed in collaboration with ARM and is extensively experimentally validated.

Our goal in developing the axiomatic mixed-size ARMv8 model is primarily to investigate JavaScript's compilation correctness. In cases where Flat's mixed-size semantics is still potentially subject to change we choose weaker behaviours, and it is possible that our model allows some mixed-size behaviours which are not allowed by Flat. As long as our model is *no stronger than* Flat, however, any compilation scheme our ARMv8 model supports will also be supported by the Flat model. We extensively validate this property experimentally, on a large corpus of tests.

In Pulte et al. [43], the uni-size axiomatic and Flat operational model were hand-proved equivalent (for uni-size input programs). Formally proving a correspondence between mixed-size Flat and a mixed-size axiomatic model would be a substantial effort in its own right: extending the axiomatic model to mixed-size accesses breaks some assumptions made by the existing proof. Extending the proof is beyond the scope of this paper where our focus is JavaScript, and further work still needs to be done in order to find axiomatic rules that are precisely equivalent to Flat. However, we believe that our approach of generalising an existing uni-size axiomatic model, combined with extensive validation, represents an important first step in solving this more general problem.

4.1 Validation

The experimental validation is based on the corpus of 11,587 existing litmus tests from prior work on ARMv8 (the majority systematically generated with `diy` [4], and including hand-written tests used in Flur et al. [21, 22]). We run the Flat model on this test suite and enumerate, for each test, the set of all behaviours allowed by Flat. We instrumented the Flat model to generate, for each such possible outcome, the candidate execution corresponding to the operational model's trace. We log the candidate executions, and feed them into the Alloy-based ARMv8 axiomatic model to ensure the *soundness* of the axiomatic model: that it allows each such Flat-allowed execution.

The litmus test suite we run contains 11,587 litmus tests. We run the tests on a Ubuntu 18.04.2 POWER9 machine (160 CPUs at 2.9GHz, 125GB ram) with no memory limit and 168h time limit. Of the 11,587 tests, 11,578 complete in Flat (2635 mixed-size and 8943 non-mixed-size), so all but 9. Of these 9, 3 are due to instructions currently unsupported by Flat, 4 running out of memory, 1 running out of time, and another test crashing with an unspecified error. For the 11,578 tests where Flat successfully completes, it generates a total of 167,014 candidate executions. We run the mixed-size Alloy-based ARMv8 axiomatic model on these and confirm that it allows every such Flat-allowed execution.

5 Alloy Verification

For the SC-DRF and ARMv8 compilation issues described in §3.2 and §3.1, we define the JavaScript and mixed-size ARMv8 models in the Alloy model checker [29], allowing us to compare the two models and investigate whether individual litmus tests are allowed by the models. This approach was first used by Wickerson et al. [53], who took existing uni-size models, written in herd [6], and automatically converted them to Alloy. In contrast, we directly transcribe the JavaScript (corrected and uncorrected) and ARMv8 models into Alloy by hand. Alloy's syntax supports arbitrary first-order predicates, so the models can be faithfully reproduced.

5.1 ARMv8 Search

We are able to use these Alloy models to test that our hand-found counter-examples are real (i.e. the execution is disallowed in JavaScript but the related execution is allowed in our ARMv8 model). In addition, following the approach of Wickerson et al. [53], we are able to use Alloy to automatically find smaller counter-examples than we were able to find manually. Our best hand-discovered counter-example for the ARMv8 violation required 8 events and 3 byte locations; Alloy finds a counter-example with 6 events, 2 byte locations.

In this search, we are looking for counter-examples to the ARMv8 compilation scheme. Such a counter-example is an execution $Exec_{\text{JS}}$ of a JavaScript program $Prog_{\text{JS}}$ that is invalid according to the JavaScript memory model, but which corresponds to an execution $Exec_{\text{ARM}}$ of a program $Prog_{\text{ARM}}$ obtained by compiling $Prog_{\text{JS}}$ to ARMv8, and where $Exec_{\text{ARM}}$ is allowed by the ARMv8 concurrency model.

To this end, we follow the approach of Wickerson et al. [53], and define a translation relation on candidate executions. Intuitively this should relate a JavaScript execution $Exec_{\text{JS}}$ with an ARM execution $Exec_{\text{ARM}}$ if $Exec_{\text{JS}}$ and $Exec_{\text{ARM}}$ are executions of the programs $Prog_{\text{JS}}$ and $Prog_{\text{ARM}}$, respectively, such that $Prog_{\text{JS}}$ compiles to $Prog_{\text{ARM}}$, and $Exec_{\text{JS}}$ and $Exec_{\text{ARM}}$ have the same observable behaviour. We define a translation relation, that:

- is compatible with the compilation scheme:
  events in $Exec_{\text{JS}}$ arising from JavaScript accesses are related to events in $Exec_{\text{ARM}}$ arising from the compiled ARMv8 accesses;
- is compatible with the program structure:
  it preserves sequenced-before edges (maps JavaScript sequenced-before edges to the matching program-order edges in ARMv8);
- preserves the observable behaviour:
  preserves reads-byte-from between $Exec_{\text{JS}}$ and $Exec_{\text{ARM}}$.

We give the event-to-event mapping of this translation below; we omit the (unsurprising) details of the mappings on relations of the candidate executions here, but give the full definition in the supplemental material [51]. The event mapping is one-to-one, except JavaScript RMW events which are implemented using a pair of load/store exclusive instructions As a minor edge-case, if the Wasm access is an unaligned non-atomic generated by a DataView, each byte of the Wasm access must be mapped to a separate single-byte ARM event of the relevant type [22].

Instructions		Events	
JavaScript	**ARMv8**	**JavaScript**	**ARMv8**
A-.load	`ldar`	$\text{R}_{\mathbf{SC}}$	R_{acq}
A-.store	`stlr`	$\text{W}_{\mathbf{SC}}$	W_{rel}
_ = b[k]	`ldr`	$\text{R}_{\mathbf{Un}}$	R
b[k] = _	`str`	$\text{W}_{\mathbf{Un}}$	W
A-.exchange	…`ldaxr/stlxr`…	$\text{RMW}_{\mathbf{SC}}$	$\text{R}_{\text{e-a}} \xrightarrow{\text{sb}} \text{W}_{\text{e-r}}$

Our Alloy counter-example search looks for a JavaScript candidate execution $Exec_{\text{JS}}$ and an ARMv8 candidate execution $Exec_{\text{ARM}}$, both well-formed, such that they are related by the translation relation, and $Exec_{\text{ARM}}$ is valid in ARMv8, but $Exec_{\text{JS}}$ invalid in JavaScript.

5.2 Finding Counter-Examples

For the uncorrected JavaScript model, we would like our search to produce counter-examples similar to Fig 6. However, naïvely searching as described above yields spurious counter-examples. An example is shown in Fig 11. This pair of executions satisfies the constraints of our search as specified so far: an invalid JavaScript execution, translation-related to a valid ARMv8 execution. JavaScript here forbids the execution, because the **rf** relation is incompatible with **tot**. However, this example is spurious, as a different choice of **tot** would make the execution allowed. Any program exhibiting this candidate execution will not be a real counter-example, because it will also exhibit the candidate execution with the correct **tot**, which is observably equivalent.

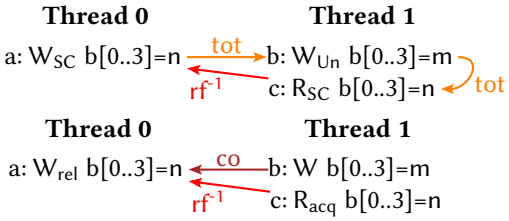


Figure 11. False counter-example from naïve search.

The problem illustrated by this example is due to the mismatch in the data of ARMv8 and JavaScript candidate executions: assuming a particular ARMv8 execution, the translation relation together with the well-formedness conditions constrains the relations of a corresponding (translation-related) JavaScript execution, except for its (existentially quantified) **tot** component. Hence the naïve counter-example

search will simply pick a "bad" **tot**, that is inconsistent with other relations of the JavaScript execution. We are only interested in counter-examples where the JavaScript execution cannot be made valid simply by permuting **tot**. Wickerson et al. [53] describe counter-example executions satisfying this requirement as having the *deadness* property.[4]

A way of guaranteeing "good" counter-examples (that are *dead*) would be specifying the search as the question: "does there *exist* a valid ARMv8 execution $Exec_{\text{ARM}}$, such that there *exists* a JavaScript execution $Exec_{\text{JS}}$, that is translation-related to $Exec_{\text{ARM}}$ and such that $Exec_{\text{JS}}$ is invalid in JavaScript *for all* total orders **tot**?" Since this Alloy search is computationally infeasible, we use the *syntactic deadness* criterion of Wickerson et al. [53]. This is a syntactic condition on candidate executions that approximates execution deadness in a way that is computationally feasible to check, but which may discard some legitimate counter-examples.

For JavaScript, any condition that guarantees that candidate executions differing only in their total-order are required to preserve $\mathbf{W_{SC}}$ **tot** W_{any} and W_{any} **tot** $\mathbf{R_{SC}}$ edges, is sufficient to guarantee deadness (we verify this in Coq, based on the model in §6). Note in particular that the "counter-example" of Fig. 11 does *not* satisfy this condition, as the **tot** edge from (a) to (b) can be inverted to create a valid execution. Defining such a search, we successfully find the counter-example in Fig. 6 of §3.1.

5.3 Bounded Compilation Correctness

With the model fixed as detailed in §3, we use Alloy to confirm that no counter-examples exist up to a bound (8 distinct events, 20 locations). This also gives us the opportunity to test proof strategies in preparation for our Coq proof of compilation scheme correctness (§6). In that proof, we must show that for any ARMv8-allowed execution a valid related JavaScript execution exists, which requires constructing a witnessing **tot** relation. We model checked our idea for this construction: making **tot** some linear extension [46] of $\mathbf{sb} \cup (\mathbf{obs} \cap (L \cup A)^2)$, where $\mathbf{obs} \cap (L \cup A)^2$ is ARM's observed-before relation restricted to release-acquire atomics (a full definition can be found in the supplementary appendix [51]) . With **tot** constrained in this way, model checking even without the syntactic deadness approximation shows the absence of compilation scheme counter examples up to the search bound.

5.4 SC-DRF Search

We are also able to automatically find counter-examples for SC-DRF in the uncorrected model. We use the same search bound, and again we must use our syntactic deadness condition to remove spurious counter-examples. We find the counter-example of Fig. 8, which is smaller than the hand-found counter-example of Watt et al. [52].

[4] Such executions are "dead" in the sense that they "cannot move around".

6 Coq Verification

We mechanise the JavaScript model, as shown in Figs. 3 and 4, in Coq.

6.1 SC-DRF

We first prove that our corrected model is SC-DRF in the sense defined in §3.2, mechanising a previous hand-proof by Watt et al. [52]:

Theorem 6.1 (internal_sc_drf)**.** *All well-formed, valid, data-race-free executions in the revised JavaScript model are sequentially consistent.*

6.2 Compilation Scheme Correctness

We now prove compilation scheme correctness, from the revised JavaScript model to our ARM model. As mentioned in §2, a limitation of this proof, not shared with our other results, is the assumption that all accesses have been generated by typed arrays (i.e. are aligned). This simplifies the proof, since unaligned ARM accesses must be split into separate bytewise events [22].

We build our proof following the style of the IMM framework [42]. As in this work, the proof proceeds by defining a "base execution" that is shared between the two models (i.e, intra-thread program order and reads-byte-from), and then showing that, for any such execution, validity in the ARM model implies validity in the JavaScript model. As an intermediate lemma, we must prove that, given an allowed ARMv8 execution, it is possible to construct a witnessing total-order relation for an allowed JavaScript execution. We achieve this proof using the construction we model-checked as part of §5.3. The initial model-checking allowed us to rapidly validate possible constructions; it would have been far more time-consuming to come up with a correct construction from scratch.

Theorem 6.2 (jsmm_compilation)**.** *The compilation scheme from the revised JavaScript model to (mixed-size) ARMv8 is correct.*

6.3 A Uni-Size Model

We can define a more standard model for JavaScript assuming uni-size accesses, where disjoint byte ranges are treated as distinct abstract locations. In the interests of space, we do not give a full definition here, but we reproduce the uni-size validity condition in Fig. 12. It is easy to see that it is a cut-down version of Fig. 4, where references to reads-byte-from are replaced with references to reads-from, and references to byte ranges are replaced with a *same-location* predicate. The **Tear-Free Reads** condition is trivially true in the uni-size case, and can therefore be removed. We mechanise our uni-size model, and a reduction from the mixed-size model to the uni-size one, proving that validity of mixed-size executions with no partial overlaps and no tearing (i.e. $\mathbf{rf}^{-1}$ being functional) is equivalent to validity in the uni-size model.

Happens-Before Consistency (1):
happens-before $\subseteq$ total-order

Happens-Before Consistency (2):
$\forall E_w E_r.\ E_w$ reads-from $E_r \longrightarrow \neg(E_r$ happens-before $E_w)$

Happens-Before Consistency (3):
$\forall (E_w, E_r) \in$ reads-from. $\quad \nexists E'_w.\ (E_w$ happens-before $E'_w) \wedge (E'_w$ happens-before $E_r) \wedge$ *same-location*(E'_w, E_r)

Sequentially Consistent Atomics:
$$\forall E_w E_r.\ E_w \text{ reads-from } E_r \ \wedge\ E_w \text{ happens-before } E_r \longrightarrow$$
$$\nexists E'_w.\ E'_w.\text{ord} = \textbf{SeqCst} \ \wedge\ E_w \text{ total-order } E'_w \ \wedge\ E'_w \text{ total-order } E_r \ \wedge$$
$$\left(\begin{array}{l} (\textit{same-location}(E'_w, E_r) \ \wedge\ E_w \text{ synchronizes-with } E_r) \\ \vee\ (\textit{same-location}(E_w, E'_w) \wedge E_w.\text{ord} = \textbf{SeqCst} \ \wedge\ E'_w \text{ happens-before } E_r) \\ \vee\ (\textit{same-location}(E'_w, E_r) \ \wedge\ E_w \text{ happens-before } E'_w \ \wedge\ E_r.\text{ord} = \textbf{SeqCst}) \end{array}\right)$$

Figure 12. Validity of uni-size JavaScript executions.

We prove compilation scheme correctness of the uni-size model to several architectures, via the Intermediate Memory Model (IMM) [38, 42]:

Theorem 6.3 (s_imm_consistent_implies_jsmm_consistent)**.** *The compilation schemes from uni-sized JavaScript to x86-TSO, POWER, RISC-V, ARMv7, and ARMv8 are correct.*

As part of this, we prove that JavaScript **Unord** accesses are no stronger than IMM **Relaxed** accesses, and JavaScript **SeqCst** accesses are no stronger than IMM **SeqCst** accesses.

6.4 Uni-Size Programs

Having defined our uni-size model and verified a reduction between executions of the mixed- and uni-size models, we must ask the question: under what restrictions does a JavaScript program produce only uni-size-reducible executions? Recalling the conditions on our proof of validity-preservation, if every SharedArrayBuffer is accessed through only a single typed array, it is guaranteed that there are no partially overlapping accesses.

However, there is still the possibility of tearing. Tearing accesses are treated as though decomposed into individual byte-wise accesses. This means that even assuming the execution has no partial overlaps, $\mathbf{rf}^{-1}$ could be non-functional (relating a read with multiple writes). JavaScript's sequential semantics guarantees that 8, 16, and 32-bit integer typed arrays will always produce tearfree accesses. However, even assuming that every typed array is one of these kinds, it is not guaranteed that $\mathbf{rf}^{-1}$ is functional.

This is because the **Tear-Free Reads** validity rule (Fig. 4) only applies to events with identical ranges. However, the **Init** event ranges over the entire memory, and thus even fully-aligned, identically ranged tearfree accesses may observe interleaving bytes from the **Init** event. Effectively, whether we model the **Init** event as tearfree or not, it will cause tearing anyway. Consider the program of Fig. 14. The 16-bit load of Thread 0 is allowed by the model to read one byte from the **Init** event and one byte from Thread 1's write, even though all loads and stores in the program are tearfree.

```
b = new Uint16Array(new SharedArrayBuffer[32])
```

Thread 0	**Thread 1**
`r = b[0];//r=0x0001`	`b[0] = 0x0101;`

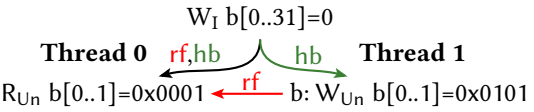


Figure 14. A tearing behaviour involving the **Init** event.

We believe this execution should not be allowed. If **Tear-Free Reads** is strengthened as follows, then $\mathbf{rf}^{-1}$ is guaranteed to be functional (assuming our typed array restrictions).

Tear-Free Reads (strong):
$$\forall E_r.\ E_r.\text{tearfree} \longrightarrow$$
$$\left|\left\{E_w \;\middle|\; \begin{array}{l} E_w \text{ reads-from } E_r \wedge E_w.\text{tearfree} \wedge \\ (\text{range}_w(E_w) = \text{range}_r(E_r) \vee E_w.\text{ord} = \textbf{I}) \end{array}\right\}\right| \leq 1$$

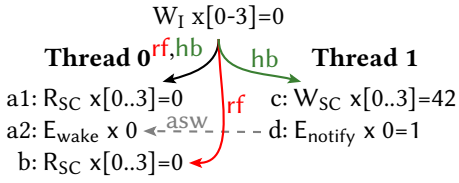
This condition intuitively seems like it should hold, and we continue to investigate whether it can be officially adopted. Note that our uni-size compilation result applies even without the revised **Tear-Free Reads** condition: the uni-size model is a *stronger* model for JavaScript programs with no partial overlaps, and nevertheless supported by the compilation schemes.

7 Atomics.wait/Atomics.notify

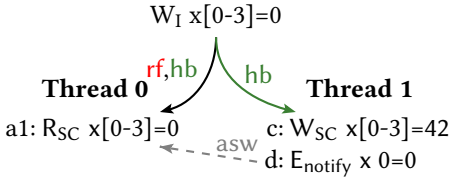
Beyond the fragment of the language that just involves memory accesses, JavaScript defines the thread synchronization operations `Atomics.wait` and `Atomics.notify`. These operations are explained by way of an example program (Fig. 13a). All operations are to the SharedArrayBuffer in `x`. The `Atomics.wait` operation reads memory location 0, and compares the result to an expected value, 0. If the expected value does not match the read value, execution continues as normal. If the expected value matches the read value, the thread suspends execution, placing itself in a *wait queue* associated with the read location. The `Atomics.notify` operation of thread 1, to the same location, will wake all threads

```
x = new Int32Array(new SharedArrayBuffer[4]);
        Thread 0                          Thread 1
a: Atomics.wait(x, 0, 0);        c: Atomics.store(x, 0, 42);
b: r0 = Atomics.load(x, 0);      d: r1 = Atomics.notify(x, 0);
```

(a) wait/notify.



(b) The interleaving a → c → d → b



(c) The interleaving c → d → a (gets stuck)

Figure 13. These two candidate executions for Fig. 13a are forbidden if the model adds the grey edges.

in the wait queue for that location. The return value of `Atomics.notify` is the number of threads woken.

Intuitively, this program should always terminate, with the load of line (b) guaranteed to read 42. If thread 0 executes `Atomics.wait` first, it will suspend until both (c) and (d) have executed, meaning (b) will not execute until 42 has been written. Alternatively, if thread 1 executes `Atomics.notify` first, then it will already have executed line (c) and written 42 in location 0. Therefore thread 0's `Atomics.wait` should continue execution as it does not observe its expected value. However, this intuition relies on the operations providing ordering guarantees, though synchronization, which are not explicitly represented in the axiomatic memory model.

Interactions with the wait queue are specified purely using an interleaving of the thread-local semantics. The specification informally describes threads updating the wait queue as entering and leaving a lock-like "critical section". However, it does not describe how the interleaved order of entry into the critical section affects the candidate executions permitted by the axiomatic memory model. We correct this so that entering the critical section implies synchronization edges in the candidate execution to all previous exits. This is in line with the treatment of locks in C/C++11 and the *monitor lock* of the Java memory model. It also fits the informal understanding of JavaScript implementers, who reported that they currently implement lock-like synchronization for `Atomics.wait/notify` [50].

These additional synchronization edges are necessary to ensure that the axiomatic model correctly forbids intuitively disallowed executions. Fig. 13b shows an undesirable execution where (b) reads 0 even though it cannot have executed until (d) notifies (a). Similarly, in Fig. 13c, (a) reads 0, suspending, even if (d) records that there were no threads notified, meaning (c) must have already executed. Incorporating the critical section entry ordering guarantees as additional-synchronizes-with edges (given by the dashed grey lines) ensures that these executions are forbidden.

8 Related Work

As mentioned in §1.1, there is little prior work dealing with mixed-size relaxed memory models. Flur et al. [22] give mixed-size operational models for ARMv8 and POWER; Pulte et al. [43] adapt this model for a revision of the ARMv8 concurrency architecture. The mixed-size ARMv8 axiomatic model presented here is directly based on this work: generalising the ARMv8 axiomatic model [17, 43] to allow for the relaxed mixed-size behaviours described by their operational model. Flur et al. also describe an extension to C/C++11's model, adding mixed-size non-atomics, and give a sketch hand-proof that the resulting model can be correctly compiled to POWER. The model of mixed-size C/C++11 is substantially simpler than JavaScript's, since non-atomics are never allowed to race. Moreover, our verification of JavaScript compilation is machine-checked.

Watt et al. [52] describe the memory model of WebAssembly, and note that it is intended to be a superset (feature-wise) of JavaScript's. The authors do not attempt verification of their proposed compilation scheme, leaving it an open problem. The core of the WebAssembly model is inherited from JavaScript, and therefore benefits from our adopted fix.

The most closely related work to our Alloy development is that of Wickerson et al. [53]. Our counter-example generation closely follows their approach for finding compilation violations for uni-size models with Alloy. We here extend this methodology to mixed-size models, although only for JavaScript and ARMv8 specifically, while their work is designed to compare arbitrary uni-size herd [6] models, allowing them to apply their tool to several existing models.

EMME [37] is an Alloy-based tool for the (uncorrected) JavaScript memory model, primarily intended as a test oracle. The authors identify and correct some earlier issues in the model, mainly related to well-formedness of certain definitions. For example, they identify that an earlier version of the model allowed RMW events to read from themselves. Their work does not concentrate on a qualitative assessment of the model, and thus does not identify the issues we describe (§3).

Hence, while we also use Alloy, our aim is different here. We found it easier to write our own JavaScript model than to adapt their model to fit with Memalloy's approach.

Some core definitions of the JavaScript memory model are shared with the C/C++11 model of Batty et al. [9], but extended to a mixed-size context. The C/C++11 model itself has been extensively formally investigated, including the correctness (or otherwise) of compilation from C/C++11 [8, 9, 31, 42, 47, 49, 53]. Several of these works, through informing the C/C++11 compilation scheme, have influenced the compilation scheme now used by JavaScript.

9 Future Work

We invest significant effort into defining and validating a mixed-size relaxed memory model for ARMv8. We benefit from the extensive body of existing work on the ARMv8 (and the related Power) memory model. To investigate compilation to other architectures, more work is needed to define their mixed-size behaviours. Most glaringly, we lack a formal model of mixed-size x86, one of the most common target platforms for JavaScript. Moreover, our ARMv8 model sidesteps some outstanding questions about the architecture's mixed-size behaviour, by, in doubt, choosing a reasonable weak option. While sufficient to justify compilation correctness, more work may be needed to improve the fidelity of the model, so it is not weaker than necessary.

JavaScript will likely one day be extended with release/acquire atomics in the style of C/C++11. We hope to engage with the standards body and use the memory model formalisation to inform such extensions.

WebAssembly's relaxed memory model is a superset (featurewise) of JavaScript's. Our approach is a first step towards verifying WebAssembly's compilation scheme, although WebAssembly's dynamic memory growth and relaxed bounds checking semantics present significant complications.

Several formalisations exist for (fragments of) JavaScript's sequential semantics [10, 26, 41]. An executable mechanisation combining JavaScript's sequential and concurrent semantics would be valuable, possibly following the approach of Nienhuis et al. [40] for the C/C++11 concurrency model.

10 Conclusion

JavaScript is a widely used language, and it is important that its shared memory concurrency is correctly specified and verified. In this paper, we investigate specification deficiencies: violations of ARMv8 compilation, and model-internal SC-DRF. We verify in Coq that our proposed fixes are correct, a first for a mixed-size model. To that end, we develop a mixed-size ARMv8 axiomatic model. Through collaboration with the standards committee, our fixes will be included in future versions of the specification.

Acknowledgments

We thank the members of ECMA TC39 for useful discussions. We thank Lars T Hansen and Peter Sewell for their support and feedback. This work was partly supported by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems* (EP/K008528/1). This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 789108, ELVER). The first author was supported by an EPSRC DTP award (EP/N509620/1) and a Google PhD Fellowship in Programming Technology and Software Engineering. The third author was supported by JetBrains Research and RFBR (grant number 18-01-00380). The fourth author was supported by ENS Rennes. The fifth author was supported by OCaml Labs.

References

[1] Allon Adir, Hagit Attiya, and Gil Shurek. 2003. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.* 14, 5 (2003), 502–515. https://doi.org/10.1109/TPDS.2003.1199067

[2] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering — a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

[3] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*. 13–24. https://doi.org/10.1145/1481839.1481842

[4] Jade Alglave and Luc Maranget. 2017. A diy "Seven" tutorial. http://diy.inria.fr/doc/index.html.

[5] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

[6] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

[7] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307.

[8] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 509–520. https://doi.org/10.1145/2103656.2103717

[9] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[10] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 87âĂŞ100. https://doi.org/10.1145/2535838.2535876

[11] A. Boduch. 2015. *JavaScript Concurrency*. Packt Publishing. https://books.google.co.uk/books?id=_fHOjgEACAAJ

[12] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[13] Martyn Capewell. 2017. [arm64] Use acquire/release memory accesses for atomics. https://codereview.chromium.org/2760963002.

[14] Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*. 16–19. https://doi.org/10.1145/1353522.1353528

[15] Chromium. 2019. Mitigating Side-Channel Attacks. https://www.chromium.org/Home/chromium-security/ssca

[16] F. Corella, J. M. Stone, and C. M. Barton. 1993. *Technical Report RC18638: A formal specification of the PowerPC shared memory architecture*. Technical Report. IBM.

[17] Will Deacon. 2016. The ARMv8 Application Level Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01).

[18] ECMA International. 2017. ECMAScript 2017 Language Specification - SharedArrayBuffer Objects. https://www.ecma-international.org/ecma-262/8.0/#sec-sharedarraybuffer-objects

[19] ECMA International. 2019. ECMAScript 2019 Language Specification - Memory Model. https://www.ecma-international.org/ecma-262/10.0/index.html#sec-memory-model

[20] ECMA TC39. 2015. Spec: JavaScript Shared Memory, Atomics, and Locks. https://github.com/tc39/ecmascript_sharedmem/blob/master/historical/Spec_JavaScriptSharedMemoryAtomicsandLocks.pdf.

[21] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 608–621. https://doi.org/10.1145/2837614.2837615

[22] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/3009837.3009839

[23] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. 1992. Programming for Different Memory Consistency Models. *J. Parallel and Distrib. Comput.* 15 (1992), 399–407.

[24] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. 635–646. https://doi.org/10.1145/2830772.2830775

[25] Théotime Grohens and Benedikt Meurer. 2018. Improving DataView performance in V8. https://v8.dev/blog/dataview

[26] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.), Vol. 6183. Springer, 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[27] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[28] David Herman, Luke Wagner, and Alon Zakai. 2014. `asm.js`. http://asmjs.org/spec/latest

[29] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290. https://doi.org/10.1145/505145.505149

[30] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 175âĂŞ189. https://doi.org/10.1145/3009837.3009850

[31] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[32] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[33] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An axiomatic memory model for POWER multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

[34] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[35] Luc Maranget. 2019. A few specific tests. http://gallium.inria.fr/~maranget/cats7/model-aarch64/tests.html#doc

[36] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf

[37] Cristian Mattarei, Clark Barrett, Shu-yu Guo, Bradley Nelson, and Ben Smith. 2018. EMME: A Formal Tool for ECMAScript Memory Model Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 55–71.

[38] Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2019. Reconciling Event Structures with Modern Multiprocessors. (November 2019). https://arxiv.org/abs/1911.06567

[39] Mozilla. 2019. Concurrency model and Event Loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[40] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 111âĂŞ128. https://doi.org/10.1145/2983990.2983997

[41] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th*

ACM SIGPLAN Conference on Programming Language Design and Implementation (Portland, OR, USA) *(PLDI âĂŹ15)*. Association for Computing Machinery, New York, NY, USA, 346âĂŞ356. https://doi.org/10.1145/2737924.2737991

[42] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

[43] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[44] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 311–322. https://doi.org/10.1145/2254064.2254102

[45] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 175–186. https://doi.org/10.1145/1993498.1993520

[46] Edward Szpilrajn. 1930. Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae* 16, 1 (1930), 386–389. http://eudml.org/doc/212499

[47] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

[48] Stephan van Schaik, Alyssa Milburn, Sebastian ÃŰsterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *40th IEEE Symposium on Security and Privacy, S&P 2019*.

[49] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

[50] Conrad Watt. 2019. (memory model, wait/notify) Atomics.wait/notify non-SC behaviour, what is expected? https://github.com/tc39/ecma262/issues/1680

[51] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu yu Guo. 2020. Supplemental Materials. https://github.com/conrad-watt/repairing-and-mechanising-the-javascript-relaxed-memory-model

[52] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 133 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360559

[53] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 190–204. https://doi.org/10.1145/3009837.3009838

[54] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Portland, Oregon, USA) *(OOPSLA 2011)*. Association for Computing Machinery, New York, NY, USA, 301âĂŞ312. https://doi.org/10.1145/2048147.2048224